# Matching in Multi-Agent Pathfinding using M*

**Jonathan Dönszelmann**[1] , **Jesse Mulderij**[1] , **Mathijs de Weerdt**[1]

[1]TU Delft

jdonszelmann@student.tudelft.nl

{m.m.deweerdt, j.mulderij}@tudelft.nl

## Abstract

Multi agent pathfinding (*MAPF*) can be extended by giving agents teams. Agents in a team need to find the best assignment of goals to minimise the *sum of individual costs*. Such an assignment is called a matching. *M** is a complete and optimal algorithm for *MAPF* without matching. This algorithm was modified to solve problems which involve matching. To do this, two strategies are proposed called *inmatching* and *prematching*. This paper shows that *prematching* is generally preferable to *inmatching*, compares the benefits of different optimisations for *M**, and shows how well *M** stacks up against other *MAPF* solving algorithms extended to perform matching.

## Introduction

A large number of real-world situations require the planning of collisionless routes for multiple agents. For example, the routing of trains over a rail network [1], directing robots in warehouses [2], or making sure autonomous cars do not collide on the road [3]. Problems of this nature are called *Multi-agent pathfinding* problems, which in this paper will often be abbreviated to *MAPF*. Solving *MAPF* problems has been proven to be **PSPACE-hard** [4].

One algorithm to solve *MAPF* is called *M** [5]. A standard *A** algorithm as described by Standley [6] plans agents together. This means that in each timestep, the number of possible next states grows exponentially with the number of agents. Contrasting that, in *M**, when possible, agents follow an individually optimal path, and in each timestep, only the subset of agents which is part of a collision is jointly planned.

The *MAPF* problem can be extended by grouping agents into teams. Each team has a number of starts and goals. Within one team, any agent can travel to any goal. Solving this problem requires an algorithm to find out which agent should move to which goal. Such an assignment between starts and goals is called a matching. This problem is therefore named multi-agent pathfinding with matching (here abbreviated to *MAPFM*).

In this paper, first *MAPFM* will be defined, and after that possibilities will be discussed to extend *M** to solve *MAPFM* problems as well. To do this, two methods will be proposed. These two methods will be compared, both to each other, and to a number other algorithms solving *MAPFM*. As well as this comparison, a number of extensions to *M** will be considered which improve *M**'s performance. Some of these extensions can also improve *M** performance on plain *MAPF* problems.

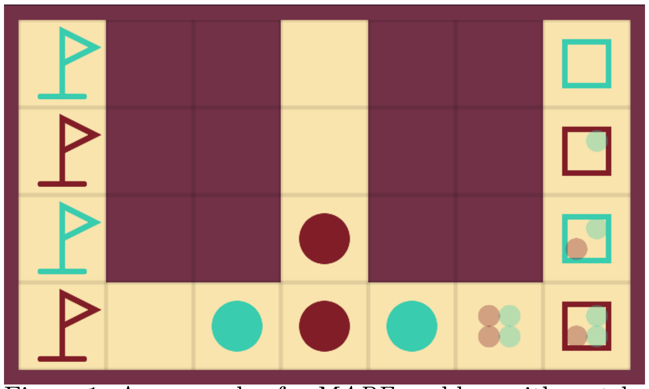


Figure 1: An example of a *MAPF* problem with matchings. Two teams (of different colours) moving from their starts (square) to their goals (flag).

## 1. Prior work

Before this research, a separate study has been performed [7] on a problem they call target-assignment and path-finding (*TAPF*). The difference between *TAPF* and *MAPFM* is that in *TAPF*, the *makespan* (the cost of the longest path) instead of the *sum of individual costs* (the cost of all paths combined) is minimised. To solve *TAPF*, [7] uses conflict based search, with a max-flow based algorithm to solve matchings within one team. It is yet unclear if it is possible to use such a max-flow based algorithm for *MAPFM*, however this is not a question that will be answered in this paper. Parallel research to this will however attempt to find an answer to this question [8].

Solving *TAPF* and *MAPFM* with other algorithms than *CBS* has not yet been explored in depth.

In contrast, a lot of research has been done finding and improving algorithms which solve *MAPF* [5], [6], [9]–[13]. But which of techniques can be efficiently extended to also solve *MAPFM* is still an open question. This paper attempts to answer this question specifically for *M**.

## 2. Definition of *MAPFM*

The definition of multi-agent pathfinding used in this paper is based on Stern's definition given in [14]. The definition is as follows:

A *MAPF* problem consists of the following elements:

$$\langle G, s, g \rangle$$

- $G$ is a graph $\langle V, E \rangle$
  - $V$ is a set of vertices
  - $E$ is a set of edges between vertices
- $s$ is a list of $k$ vertices where every $s_i$ is a starting position for an agent $a_i$
- $g$ is a list of $k$ vertices where every $g_i$ is a target position for an agent $a_i$

Although algorithms presented in this paper would work on any graph $G$, in examples and experiments, $G$ is simplified to be a 4-connected grid. In this paper, this grid is sometimes called a map, on which agents move from their starts to their goals.

This definition is now expanded to a definition of multi-agent pathfinding with matching. This expanded definition is called *MAPFM*. A definition of this new problem is as follows:

$$\langle G, s, g, sc, gc \rangle$$

- $sc$ is an array of colours $sc_i$ for each starting vertex $s_i$
- $gc$ is an array of colours $gc_i$ for each target vertex $g_i$

This definition divides all agents into teams. An agent $a_i$'s team colour is the colour of its starting location $sc_i$. In total there are $K$ teams $k_1 \dots k_n$. $K$ is equal to the number of different colours in *sc* and *gc*.

In *MAPFM*, a goal state is a state in which each agent is on a goal location in $g$, such that the colour of every agent is equal to the colour of the goal they are on.

*MAPFM* disallows vertex and edge conflicts as described in [14]. This means that two agents $a_i$ and $a_j$ cannot be at the same vertex $v$ or edge $e$ at a timestep $t$.

*MAPFM* is an *optimisation problem.* A solution to this problem has a cost $c$ which needs to be minimised. The cost of an agent $a_i$'s path is defined to be the number of timesteps it takes for an agent to reach its goal and never leave it again. This means that it's possible for an agent to reach the goal in an earlier timestep, and leave it again later to let another agent pass.

To find the cost of a solution for all agents, [14] presents two methods. In this paper, the *sum of individual costs* is minimised. The cost of a solution is thus the sum of the cost of all agents their paths.

## 3. A description of *M**

*M** [5] is a pathfinding algorithms which searches a search space, similar to *A**, but is specifically designed to work with multiple agents. In *M**, the search space consists of the positions of all the agents on a grid. To use *A** for multiple agents, the planning of paths for all agents needs to be coupled together to make sure no states are expanded in which collisions occur.

Unlike *A**, in *M** the planning of agents is initially not coupled. Instead, *M** assumes that the optimal path for an agent to their goal, does not contain any collisions. As long as this is true, the planning of agents is separated.

However, the assumption that all optimal paths are collisionless obviously does not always hold. Therefore, each state in *M** holds, apart from the positions of each agent, also two sets called the *collision set* and then *backpropagation set.* When *M** detects that a state contains collisions, the algorithm does not continue expanding this state. It instead uses information stored in these two sets to backtrack and find the shortest route around these collisions. Agents associated in the collision temporarily plan routes in a coupled fashion.

After collisions, *M** plans agents independently again according to their individually optimal path. It does however record information about the previous collisions (in these *backtracking-* and *collision sets*). This is to make sure that when in the future another collision occurs, it can either resolve this locally or backtrack back to the previous collision to plan it differently to potentially avoid this new collision altogether.

[5] proves that *M** provides optimal solutions to *MAPF*.

## 4. *M** and matching

To add matching to *M**, this paper proposes two options which are proposed to be named "inmatching" and "prematching". In this section, both are explained and their advantages and disadvantages are discussed.

### 4.1. Inmatching

*Inmatching* is the process of performing matching as a part of the pathfinding alorithm that is used. To understand it, it is useful to first look at inmatching in *A**. With *A**, the expansion of a state are all combinations of moves for all agents. *A** searches through the search space, until the goal state is removed from the frontier. Given an admissible heuristic *A** will guarantee that following the children of this first goal state gives a shortest path.

With *inmatching*, there is not one goal state. Instead any state in which all agents are on a goal of their own colour is considered a goal state. This means there is more than one goal state. To keep the heuristic admissible, the distance to the nearest goal state is used as a heuristic.

Inmatching can similarly be applied to *M**. There is however something that is important to keep in mind. One of the core ideas of *M** is that agents, when not colliding, follow an individually optimal path to their goal. With *inmatching*, every goal in the team of the agent is a valid goal. Therefore, to preserve optimality, agents should consider optimal paths to each of the goals in the team.

The implication of this is that state, even when agents are not colliding, can expand into an exponential number of new states. An upper bound for the size of the expansion is as follows:

$$\prod_{n=1}^{K} \text{number of goals}(k_n)$$

*M** using *inmatching* will from now on be abbreviated with *imM**

### 4.2. Prematching

Alternatively, there is *prematching*. With *prematching* the *MAPFM* problem is transformed in a number of *MAPF* problems. Each possible matching is calculated in advance, and normal *M** as described by [5] is performed on each matching. Algorithm 1 shows how this is done.

*M** using *prematching* will from now on be abbreviated with *pmM**

```
Algorithm 1: prematch $M*$
  Result: Find the matching with smallest cost
  foreach $m \leftarrow$ find all matchings(starts, goals) do
    | $S(i) \leftarrow \text{mstar}(starts, m)$ {Evaluate with $M*$}
  end
  return min(S) {by calculated cost}
```

## 5. Extensions to *M**

*M** can be improved upon in various ways. Some of these extensions are only applicable to matching *M**, while others improve *M** itself.

In this section these extensions are explained, and their effects are presented.

### 5.1. Recursive M*

With regular *M**, if on a timestep $t$, agent $a_i$ collides with agent $a_j$ and agent $a_k$ collides with agent $a_l$, all these 4 agents will be treated as colliding agents and their routes are planned together. Even though what's actually happening is that there are two, disjoint sets of agents which are colliding. For this situation, [5] provides a technique called recursive *M** (abbreviated to *rM**). With *rM**, the algorithm recursively runs mstar on these disjoint colliding subsets.

[5] found that using *rM** makes it so the computational cost grows sub-exponentially with the number of agents on average.

### 5.2. Precomputed paths and heuristics

A part of solving *M** is finding the individually optimal path for each agent. This can be done with conventional pathfinding algorithms such as *A**. But to avoid repeated recalculation, to first create a lookup table of the distance to each goal, for every open square on the map.

When using *prematching*, the same goals and starts are (although in a different permutation) reused multiple times. The lookup table can remain the same every time.

Using this lookup table, the heuristic *M** can be improved as well. Instead of using the *euclidean-* or *Manhattan distance* to the goal, the lookup table can be used to find the exact distance. The heuristic found with this approach also takes the obstacles on the map into account.

## 5.3. Operator decomposition

In [6], Standley describes a technique called operator decomposition (*OD*). With operator decomposition the number of expanded nodes is divided by $n$ (where $n$ is the number of agents), while the search depth is multiplied by $n$.

To do this, expansions can be partial or complete. Each time a state is expanded, only the moves of one agent are expanded and put back in the queue. Only when this partially expanded node comes to the top of the queue the moves of another agent are considered. Once all agents in a state did their expansion, the state is said to be complete again.

The queue now contains both partial and complete states. This provides the search algorithm with more granularity in prioritising moves, which in turn can mean a considerable improvement in runtime.

**M*** can also benefit from operator decomposition as explained in [15].

## 5.4. Pruning of matchings

*pmM** as previously described, usually has to evaluate every matching of every team. However, it is possible to discard some matchings without evaluating them with *M** at all, by using heuristics.

To do this, first calculate the heuristic (i.e. the sum of distances between starts and goals) of the initial state of every matching. Because this heuristic is admissible, it represents a lower bound for the cost of this matching.

Now, before every evaluation of a matching $m$, its heuristic is first checked against the best matching $m_{best}$ found so far. If heuristic$(m) \geq cost(m_{best})$, there is no need to evaluate $m$ and $m$ can be pruned.

### 5.4.1. Sorting

To take maximum advantage of pruning, matchings can be sorted based on their heuristic. Making sure the matching with the lowest heuristic is evaluated first can increase chances that later matchings are pruned. Algorithm 2 shows how pruning with sorting works.

**Algorithm 2:** prematch *M** with pruning

```
Result: Find the matching with smallest cost
        without evaluating every matching.
$m_{best} \leftarrow \varnothing$
$M \leftarrow$ find all matchings$(starts, goals)$;
sort$(M)$ {on heuristic; ascending}
foreach $m \leftarrow M$ do
    if $heuristic(m) < cost(m_{best})$ then
        $s \leftarrow$ mstar$(starts, m)$;
        $m_{best} \leftarrow min(m_{best}, s)$;
    end
end
```

## 6. Results

To evaluate the performance of matching *M**, a number of experiments were performed. For each of these experiments, the algorithm used was written in **python 3.9** and benchmarks were run on a virtualized system with a 12 core 12 thread Intel Xeon E5-2683 running at 2GHz, which has 8Gb of RAM.

## 6.1. Matching strategies

In section 4, two strategies were proposed to add matching to *M**. To assess their performance, both algorithms were tasked with solving a set of randomly generated maps. Both algorithms solved the same set of maps, and the locations of starts and goals on these maps are uniformly distributed.

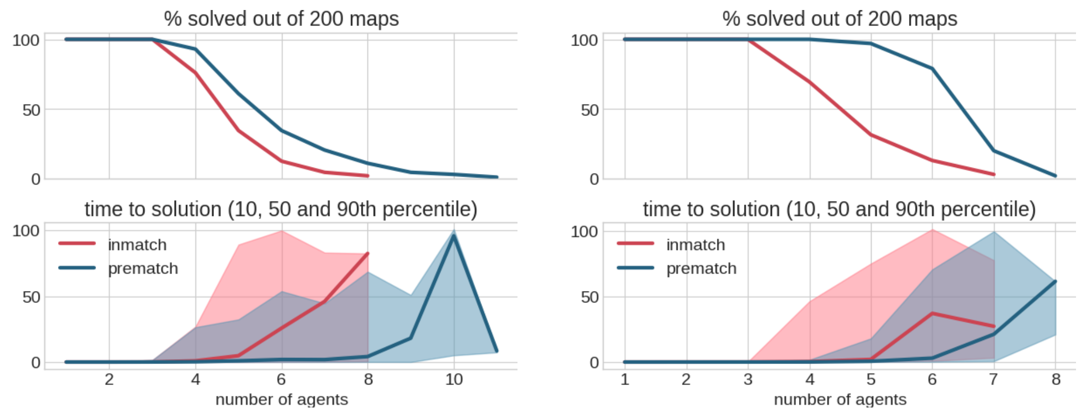
Out of a set of 200 of these randomly generated maps, the percentage of maps that the algorithm could solve in two minutes was recorded every time. Figure 2 shows the results of this experiment with differing amounts of walls and teams.

## 6.2. Extensions to *M**

Various extensions have been proposed in section 5. To prove their benefits, a comparison has been made between *M** without any extensions, all extensions individually, and finally all extensions together.

A choice was made to only use *prematching* as a matching strategy here, as it has proved to be a superior method in practice, as shown in section 5.

Results were gathered following the same method as described in section 6.1, using random maps with uniformly distributed goals and starts. These results can be found in figure 4.



(a) 3 teams. Left: 75% wall, right: 25%wall

(b) 1 team. Left: 75% wall, right: 25%wall

Figure 2: Percentage of maps solved in 2 minutes out of 200 random 20x20 maps

## 7. Comparison to other algorithms

This research is part of a set of parallel studies on how to extend a collection of *MAPF* algorithms to give them the ability to do matching. These are

- Extended partial expansion *A** (*EPA**) [10]
- *A** with operator decomposition and independence detection (*A*+OD+ID*) [6]
- Increasing cost tree search (ICTS) [13]
- Conflict based search (CBS) [12]

to give them the ability to do matching. For this paper only *M** was implemented, while data comparing *M** to other algorithms was shared between these parallel studies.

To ensure a fair comparison, each of the algorithms ran the exact same set of benchmarks on the same computer.

## 8. Future work

## 9. Reproducibility

Results in this paper have been generated using an implementation of *M** made in python specifically for this research. The code for this is publicly available on

Figure 3: Average expansion size on the same maps as used in figure 2b. Left: 75% wall, right: 25%wall

(a) 3 teams. Left: 75% wall, right: 25%wall

(b) 1 team. Left: 75% wall, right: 25%wall

Figure 4: Percentage of maps solved in 2 minutes out of 200 random 20x20 maps

github at https://github.com/jonay2000/mstar doubly licensed under the Apache and MIT licenses. Reports of bugs and new additions to this codebase are always welcomed.

## 10. Conclusion

## References

[1] J. Mulderij, B. Huisman, D. Tönissen, K. van der Linden, and M. de Weerdt, "Train unit shunting and servicing: A real-life application of multi-agent path finding," *arXiv preprint arXiv:2006.10422*, 2020.

[2] J. Li, A. Tinka, S. Kiesel, J. W. Durham, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding in large-scale warehouses," *arXiv preprint arXiv:2005.07371*, 2020.

[3] A. Mahdavi and M. Carvalho, "Distributed coordination of autonomous guided vehicles in multi-agent systems with shared resources," in *2019 SoutheastCon*, IEEE, 2019, pp. 1–7.

[4] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; pspace-hardness of the" warehouseman's problem"," *The International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[5] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *2011 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2011, pp. 3260–3267.

[6] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, 2010.

[7] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," *arXiv preprint arXiv:1612.05693*, 2016.

[8] R. Baauw, M. de Weerdt, and J. Mulderij, "Todo," 2021.

[9] E. Lam, P. Le Bodic, D. D. Harabor, and P. J. Stuckey, "Branch-and-cut-and-price for multi-agent pathfinding.," in *IJCAI*, 2019, pp. 1289–1296.

[10] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, and J. Schaeffer, "Enhanced partial expansion a*," *Journal of Artificial Intelligence Research*, vol. 50, pp. 141–187, 2014, ISSN: 10769757. DOI: `10.1613/jair.4171`.

(a) 3 teams. Left: 75% wall, right: 25%wall

(b) 1 team. Left: 75% wall, right: 25%wall

(c) 1 team. Left: 75% wall, right: 25%wall

Figure 5: TODO

[11] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. C. Holte, "Partial-expansion a* with selective node generation," *Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012*, pp. 180–181, 2012.

[12] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015.

[13] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2013.

[14] R. Stern, N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. Kumar, *et al.*, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *arXiv preprint arXiv:1906.08291*, 2019.

[15] C. Ferner, G. Wagner, and H. Choset, "Odrm* optimal multirobot path planning in low dimensional search spaces," in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 3854–3859.